NUMERICS USER GUIDE

R. VERFÜRTH

CONTENTS

1. Introduction 1
2. Installing `Scilab` 2
3. Installing `Numerics` 2
4. Using `Numerics` 2
4.1. Data structures 2
4.2. Structure of `Numerics` functions 4
4.3. Start 5
4.4. Help 5
4.5. Interpolation 5
4.6. Numerical integration 8
4.7. Nonlinear equations 8
4.8. Iterative solvers for linear systems of equations 9
4.9. Eigenvalue problems 11
4.10. Initial value problems for odes: fixed step-size methods 12
4.11. Initial value problems for odes: variable step-size methods 14
4.12. Boundary value problems for odes 17
4.13. Finite difference methods for pdes 17
4.14. Linear optimization 19
4.15. Discrete optimization 20
4.16. Nonlinear optimization 23
References 25
Index 27

1. INTRODUCTION

`Numerics` is a library of `Scilab` functions implementing many of the algorithms presented in our courses on numerical analysis for mathematicians and engineers [5, 6, 7, 8, 9, 10]. It covers the following areas:

- interpolation,
- numerical integration,
- nonlinear equations,

Date: January 5, 2012.

- iterative solvers for linear systems of equations,
- eigenvalue problems,
- initial value problems for odes,
- boundary value problems for odes,
- finite difference methods for pdes,
- linear optimization,
- integer optimization,
- nonlinear optimization.

A detailed description of `Numerics` is given in section 4 below. In sections 2 and 3 we first explain how to install `Scilab` and `Numerics`, respectively.

2. Installing Scilab

`Scilab` is a numerical programming environment comparable to `Matlab` with a similar syntax. It was developed by `INRIA`, France and is currently maintained by the `Digiteo`-foundation in collaboration with `INRIA`. Contrary to `Matlab`, `Scilab` is completely free of charge. There are versions for the operating systems `Mac OsX`, `Unix` and `Windows`. To use `Numerics` you first have to get the appropriate version of `Scilab` from

http://www.scilab.org/

and install it on your computer following the instructions of that website. Useful introductions to `Scilab` are e.g. [1, 2]; [4] may be of particular interest for those familiar with `Matlab`.

3. Installing Numerics

To install `Numerics` click the link `Numerics.zip` on

http://www.rub.de/num1/softwareE.html

or

http://www.rub.de/num1/software.html.

This installs a zip-archive `Numerics.zip` on your computer. Unpack the zip-archive and put the created folder `Numerics` at any place on your computer that suits you. Now you can start `Numerics` as described in section 4.3.

4. Using Numerics

4.1. Data structures.

We refer to [1, 2, 4] for a detailed introduction to `Scilab`. Here, we briefly present those data structures of `Scilab` that are used for input arguments of `Numerics` functions.

Boolean: Boolean arguments are transferred to a `Numerics` function by either directly entering `%t` for the value `true` or `%f` for the value `false` or by creating an own boolean variable `myboolean` by typing `myboolean = %t` or `myboolean = %f` in the `Scilab` console and then

calling the `Numerics` function with `myboolean` as argument.
String: Strings are enclosed by quotation marks. They are transferred to a `Numerics` function by either directly entering `"name"` for the string value `name` or by creating an own string variable `mystring` by typing `mystring = "name"` in the `Scilab` console and then calling the `Numerics` function with `mystring` as argument.
Number: Numbers may be integers like 2 and -3 or fixed point reals like 3.14 and -2.87 or floating point reals like $0.578\mathtt{D}4$, $-0.689\mathtt{D}5$, $0.203\mathtt{D}-7$, and $-0.1763\mathtt{D}-5$. They are transferred to a `Numerics` function by either directly entering the numerical value or by creating an own variable `mynumber` by typing `mynumber = 2` or similar in the `Scilab` console and then calling the `Numerics` function with `mynumber` as argument.
Vector / Matrix: A vector is either an $m \times 1$ matrix for a column vector or a $1 \times n$ matrix for a row vector. Matrices are created by typing the appropriate sequence of numbers in lexicographic order from top left to bottom right in brackets with columns separated by blanks or commas and rows separated by semicolons. Thus the matrices

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

are represented by

$$[1\ 2\ 3; 4\ 5\ 6] \quad \text{and} \quad [1, 2; 3, 4; 5, 6].$$

A' is the transpose of the matrix A; $A * B$ is the product of the matrices A and B. `ones`(m, n) and `zeros`(m, n) create $m \times n$ matrices with all elements equal to 1 and 0, respectively; `eye`(n, n) creates the $n \times n$ identity matrix. Matrices are transferred to a `Numerics` function by either directly entering the matrix or by creating an own matrix-valued variable `mymatrix` and then calling the `Numerics` function with `mymatrix` as argument.
Sparse matrix: `Scilab` has a built-in data structure for sparse matrices which differs from the corresponding structure of `Matlab`. A sparse matrix with n non-zero entries is stored in an $n \times 1$ column vector containing the non-zero entries and an $n \times 2$ matrix containing the row and column indices of the non-zero entries. Suppose you want to store the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & -1 & 0 \end{pmatrix}$$

in sparse format. Then you must enter in the `Scilab` console (without the full stop at the end!)

$$\begin{aligned} \texttt{myentries} &= [1; 2; 3; 4; -1] \\ \texttt{myindices} &= [1, 1; 1, 4; 2, 3; 4, 1; 4, 3] \\ \texttt{mysparsematrix} &= \texttt{sparse}(\texttt{myindices}, \texttt{myentries}). \end{aligned}$$

Note that the input

$$\begin{aligned} \texttt{myoentries} &= [-1;1;2;3;4] \\ \texttt{myoindices} &= [4,3;1,1;1,4;2,3;4,1] \\ \texttt{myosparsematrix} &= \texttt{sparse}(\texttt{myoindices},\texttt{myoentries}) \end{aligned}$$

represents the same matrix. This matrix in sparse format is transferred to a `Numerics` function by using `mysparsematrix` or `myosparsematrix` or an equivalent construct as argument.
Function: Functions, user-defined or built-in ones, can act as arguments of `Numerics` functions. To clarify this, consider two examples. Suppose first that you want to interpolate the built-in exponential function in 10 equidistant points on the interval $[-1,1]$. Then you simply enter

`lagrange`(`exp`,10)

in the `Scilab` console. This invokes the `Numerics` function `lagrange` with the built-in exponential function `exp` and the number 10 as arguments. Suppose next that you want to interpolate the function $\frac{x}{2+x^2}$ in 10 equidistant points on the interval $[-1,1]$. Then you first have to create the corresponding function by entering

```
function y = myfunction(x)
y = x/(2 + x * x)
endfunction
```

in the `Scilab` console. Then you may start the interpolation by typing (without the full stop at the end!)

`lagrange`(`myfunction`,10).

4.2. **Structure of `Numerics` functions.** `Numerics` functions never have a return value, they only print the required time for the calculation. In addition, depending on their arguments, they may print and eventually plot results. All functions have a certain number of mandatory and optional arguments. Mandatory arguments must be specified by the user. Optional arguments may be omitted, corresponding default values will be used then. Note that arguments are identified by order. Consequently you must enter a void argument [] for all those optional arguments that you don't want to change and that come prior to an optional argument which you want to change. As an example consider the function `lagrange`. It requires a function as mandatory first argument, the number of interpolation points as mandatory second argument, the left endpoint of the interpolation interval as optional third argument with default value -1, the right endpoint of the interpolation interval as optional fourth argument with default value 1 and further optional arguments which we do not need for this example. Then the inputs

```
lagrange(exp,10)
lagrange(exp,10,-2)
lagrange(exp,10,[],2)
lagrange(exp,10,-4,3)
```

yield the Lagrange interpolation of the built-in function `exp` in 10 equidistant points on the intervals $[-1, 1]$, $[-2, 1]$, $[-1, 2]$ and $[-4, 3]$, respectively.

4.3. **Start.** Before using any `Numerics` function you must make it known to `Scilab` by first entering

```
mylib = lib(path)
```

to the console. Here, `mylib` may be any other name different from the name of any `Numerics` function, any built-in function and any of your user-defined functions. `path` is a string giving the full path to the `Numerics` library. Suppose for example that you are using `Mac OsX`, that `Gargantua` is your user name and that you have put the `Numerics` folder into the subfolder `Pantagruel` of your documents folder. Then the above command must read (without the full stop at the end!)

```
mylib =
lib("/users/Gargantua/documents/Pantagruel/Numerics").
```

If everything is correctly installed, `Scilab` returns the names of *all* functions in the `Numerics` library. This includes auxiliary functions which are not user-relevant and which will not be described below.

4.4. **Help.** `Numerics` comes with a help function `numerics_help` which may be used with any number of arguments including an empty argument list. Typing `numerics_help()` yields a list of all user-relevant functions for which help is available. When specifying arguments, these must be strings with the names of `Numerics` functions. Thus typing

```
numerics_help("newton","secant_rule","regula_falsi")
```

provides information on the functions `newton`, `secant_rule` and `regula_falsi`.

4.5. **Interpolation.** `Numerics` provides the functions

`neville`, `lagrange`, `hermite`, `cubic_spline`, `goertzel`

for interpolation. These functions can be used in two different ways which are determined by the type of their arguments:

- interpolation of a function,
- interpolation of discrete data.

The function `neville` realizes interpolation by Neville's scheme (see [5, Algorithmus I.2.7]). When interpolating a given function the user must provide the following arguments the first three being mandatory:

`y:` scalar, vector or matrix of points in which the interpolation polynomial should be evaluated,

`f:` function that should be interpolated,

`n:` number n of interpolation points (at least 2),
`a:` first interpolation point a, default is -1,
`b:` last interpolation point b, default is 1,
`t:` spacing of interpolation points,
`"e"`: equidistant points $a + (b-a)\frac{i-1}{n-1}$, $1 \le i \le n$,
`"c"`: Čebysev points $a + \frac{1}{2}(b-a)(1-\cos(\pi\frac{i-1}{n-1}))$, $1 \le i \le n$,
default is `"e"`.

When interpolating discrete data the user must provide the following three mandatory arguments:

`y:` scalar, vector or matrix of points in which the interpolation polynomial should be evaluated,
`f:` vector of function values at the interpolation points (at least 2),
`x:` vector of interpolation points (at least 2).

Note that both vectors `f` and `x` must have the same dimensions. In both cases `neville` prints the vectors of the evaluation points and of the corresponding values of the interpolation polynomial.
The function `lagrange` performs Lagrange interpolation using divided differences and Newton's scheme (see [5, Algorithmus I.2.11, I.2.13], [8, Algorithmus III.2, III.3]). When interpolating a given function the user must provide the following arguments the first two being mandatory:

`f:` function that should be interpolated,
`n:` number n of interpolation points (at least 2),
`a:` first interpolation point a, default is -1,
`b:` last interpolation point b, default is 1,
`t:` spacing of interpolation points,
`"e"`: equidistant points $a + (b-a)\frac{i-1}{n-1}$, $1 \le i \le n$,
`"c"`: Čebysev points $a + \frac{1}{2}(b-a)(1-\cos(\pi\frac{i-1}{n-1}))$, $1 \le i \le n$,
default is `"e"`
`ap:` left endpoint α of plot interval, default is a,
`bp:` right endpoint β of plot interval, default is b,
`np:` number k of plot points, default is 101.

When interpolating discrete data the role of the first two arguments changes: `f` now is the vector of the function values that should be interpolated and `n` is replaced by the vector `x` of the corresponding abscissa. Both vectors must have the same dimensions. The function `lagrange` prints the maximal interpolation error and plots the interpolation polynomial, the original function and the error by drawing a piecewise linear interpolation of these functions at k equidistant points on the interval $[\alpha, \beta]$.
The function `hermite` performs Hermite interpolation using divided differences and Newton's scheme (see [8, §III.4]). When interpolating a given function the user must provide the following arguments the first three being mandatory:

`f:` function that should be interpolated,
`df:` derivative of the function that should be interpolated,
`n:` number n of interpolation points (at least 2),
`a:` first interpolation point a, default is -1,
`b:` last interpolation point b, default is 1,
`t:` spacing of interpolation points,
`"e"`: equidistant points $a + (b-a)\frac{i-1}{n-1}$, $1 \le i \le n$,
`"c"`: Čebysev points $a + \frac{1}{2}(b-a)(1-\cos(\pi\frac{i-1}{n-1}))$, $1 \le i \le n$,
default is `"e"`
`ap:` left endpoint α of plot interval, default is a,
`bp:` right endpoint β of plot interval, default is b,
`np:` number k of plot points, default is 101.

When interpolating discrete data the role of the first three arguments changes: `f` and `df` now are the vectors of the function values and corresponding derivatives, resp. that should be interpolated and `n` is replaced by the vector `x` of the corresponding abscissa. These three vectors must have the same dimensions. The function `hermite` prints the maximal interpolation error and plots the interpolation polynomial, the original function and the error in the same way as the function `lagrange`.
The function `cubic_spline` realizes interpolation by natural cubic splines (see [5, Satz I.3.4], [8, Algorithmus III.8]). When interpolating a given function the user must provide the following arguments the first two being mandatory:

`f:` function that should be interpolated,
`n:` number n of interpolation points (at least 2),
`a:` first interpolation point a, default is -1,
`b:` last interpolation point b, default is 1,
`t:` spacing of interpolation points,
`"e"`: equidistant points $a + (b-a)\frac{i-1}{n-1}$, $1 \le i \le n$,
`"c"`: Čebysev points $a + \frac{1}{2}(b-a)(1-\cos(\pi\frac{i-1}{n-1}))$, $1 \le i \le n$,
default is `"e"`
`np:` number k of plot points per subinterval, default is 10.

When interpolating discrete data the role of the first two arguments changes: `f` now is the vector of the function values that should be interpolated and `n` is replaced by the vector `x` of the corresponding abscissa. Both vectors must have the same dimensions. The function `cubic_spline` prints the maximal interpolation error and plots the interpolation polynomial, the original function and the error by drawing a piecewise linear interpolation of these functions at k equidistant points in each subinterval of $[a, b]$.
The function `goertzel` performs trigonometric interpolation using the Goertzel scheme (see [5, Algorithmus I.5.3]). When interpolating a given function the user must provide the following arguments the first one being mandatory:

`f:` function f that should be interpolated,
`n:` interpolate f at the points $\frac{2\pi\ell}{2n+1}$, $0 \le \ell \le 2n$,
`np:` number k of plot points, default is 101.

When interpolating discrete data the role of the first argument changes: `f` now is the vector of the function values that should be interpolated. The function `goertzel` prints the maximal interpolation error and plots the interpolation polynomial, the original function and the error by drawing a piecewise linear interpolation of these functions at k equidistant points on the interval $[0, 2\pi]$.

4.6. **Numerical integration.** `Numerics` provides the functions

`midpoint_rule`, `trapezoidal_rule`, `simpson_rule`,
`gauss_rule`, `romberg`

for numerical integration.
The functions `midpoint_rule`, `trapezoidal_rule`, `simpson_rule` implement the corresponding composite quadrature formulae (see [5, Beispiel II.2.2(3)], [8, §IV.2]). They all have the following arguments the first one being mandatory and print the computed approximate value of the integral:

`f:` function that should be integrated,
`a:` left endpoint of integration interval, default is 0,
`b:` right endpoint of integration interval, default is 1,
`nc:` initial number n of subintervals, default is 1,
`nr:` number k of refinement steps, default is 0,
in the ℓ-th refinement step, $0 \le \ell \le k$, the integration interval is split into $n2^\ell$ subintervals.

The function `gauss_rule` implements a composite Gauß quadrature formula with m interior Gauß points and order $2m - 1$ (see [5, §II.3], [8, §IV.4]). The value of m is passed as optional sixth argument `ng`. The value of `ng` may be 2, 3 or 4 with 4 being the default value.
The function `romberg` realizes the Romberg scheme (see [5, Definition II.4.6], [8, §IV.6]). It has six arguments the first five being the same as those of `midpoint_rule`. The optional sixth argument `ne` determines the number of columns of the Romberg scheme. Its default value is 5 and it should be at least 2 and at most $\min\{5, \texttt{nr}\}$.

4.7. **Nonlinear equations.** `Numerics` provides the functions

`newton`, `secant_rule`, `regula_falsi`

for the solution of nonlinear (systems of) equations.
The function `newton` implements Newton's method with damping for systems of nonlinear equations $f(x) = 0$ in $\mathbb{R}^n$ (see [5, Definition III.2.1], [8, §§II.3, II.5]). It has the following arguments the first three being mandatory:

`x:` initial value,
`f:` function f,

`df:` derivative Df of f,
`pr:` if `%t` results are printed in each iteration, default is `%f`,
`nod:` if `%t` damping is switched off, default is `%f` i.e. damping is performed,
`tol:` error tolerance ε, iteration is stopped once $\sqrt{\frac{1}{n}f(x)\cdot f(x)} \leq \varepsilon$, default is 1.0D−8,
`nmax:` maximal number of iterations, default is 10.

The function `newton` prints the required number of iterations, the final residual $\sqrt{\frac{1}{n}f(x)\cdot f(x)}$ and the approximate solution x.
The function `secant_rule` realizes the secant rule for scalar nonlinear equations $f(x) = 0$ in $\mathbb{R}$ (see [5, Algorithmus III.3.1], [8, Algorithmus II.3]). It has the following arguments the first three being mandatory:

`x1:` first initial value x_0,
`x2:` second initial value x_1,
`f:` function f,
`pr:` if `%t` results are printed in each iteration, default is `%f`,
`tol:` error tolerance ε, iteration is stopped once $|f(x)| \leq \varepsilon$, default is 1.0D−8,
`nmax:` maximal number of iterations, default is 10.

The function `secant_rule` prints the required number of iterations, the final residual $|f(x)|$ and the approximate solution x.
The function `regula_falsi` implements the regula falsi for scalar nonlinear equations $f(x) = 0$ in $\mathbb{R}$ (see [5, Algorithmus III.3.2]). It has the same arguments as `secant_rule` and produces the same output but the function values $f(x_0)$ and $f(x_1)$ of the two initial values must now be of different sign.

4.8. **Iterative solvers for linear systems of equations.** `Numerics` provides the functions

`richardson`, `jacobi`, `gauss_seidel`, `ssor`,
`cg`, `pcg_ssor`, `bicg_stab`, `pbicg_stab_ssor`,
`compare_solvers`

for the iterative solution of linear systems of equations $Ax = b$. With the exception of `compare_solvers` all functions print intermediate and final results depending on their argument `prl` and plot the relative residuals $\frac{\|Ax_n-b\|}{\|Ax_0-b\|}$, convergence rates $\frac{\|Ax_n-b\|}{\|Ax_{n-1}-b\|}$ and mean convergence rates $\sqrt[n]{\frac{\|Ax_n-b\|}{\|Ax_0-b\|}}$ over the number n of iterations.
The functions `richardson`, `jacobi`, `gauss_seidel` and `ssor` implement the corresponding classical stationary iterative methods (see [5, Beispiel IV6.3, Algorithmus IV.7.14], [10, §§II.2.2–II.2.5]). They all have the following arguments the first two being mandatory:

`A:` matrix A,

`b:` right-hand side vector b,
`x:` : initial guess x_0, default is $x_0 = 0$,
`om:` damping parameter, default is $\frac{1}{n}\sum_{ij}|A_{ij}|$ for `richardson`, 1 for `jacobi` and `gauss_seidel` and 1.5 for `ssor`,
`prl:` print level, default is 0
(0) only print required number of iterations n, final residual $\|Ax_n - b\|$ and mean convergence rate $\sqrt[n]{\frac{\|Ax_n-b\|}{\|Ax_0-b\|}}$,
(1) print actual iteration number n, actual residual $\|Ax_n - b\|$ and actual convergence rate $\frac{\|Ax_n-b\|}{\|Ax_{n-1}-b\|}$,
(2) same as 0, additionally print the solution vector x_n,
(3) same as 1, additionally print the current vector x_n,
`tol:` tolerance ε, default is 1.0D−8,
`nmax:` maximal number N of iterations, default is 100,
iteration is stopped if $\frac{\|Ax_n-b\|}{\|Ax_0-b\|} \le \varepsilon$ or $n = N$.

The functions `cg` and `pcg_ssor` realize the conjugate gradient algorithm (see [5, Algorithmus IV.7.7], [9, Algorithmus VI.4.1], [10, §III.3.2]) and the preconditioned conjugate gradient algorithm with SSOR-preconditioning (see [5, Algorithmen IV.7.10, IV.7.14], [9, Algorithmen VI.4.2, VI.4.3], [10, §§III.3.3, III.3.4]). `cg` has the same arguments as `richardson` except the missing relaxation parameter `om`. `pcg_ssor` has the same arguments as `richardson` with the default value 1.5 for the relaxation parameter `om`. Notice that for both algorithms the matrix A must by symmetric positive definite.
The functions `bicg_stab` and `pbicg_stab_ssor` implement the stabilized bi-conjugate gradient algorithm (see [9, Algorithmus VI.7.1], [10, Algorithm III.5.1]) and the preconditioned stabilized bi-conjugate gradient algorithm with SSOR-preconditioning. They have the same arguments as their 'symmetric' counterparts `cg` and `pcg_ssor` up to an additional optional final argument `nmaxrestart` which limits the maximal number of restarts with the default value being 10.
The function `compare_solvers` compares various direct and iterative solvers for the solution of the linear system of equations arising from the five-point difference discretization of the Poisson equation with homogeneous Dirichlet boundary conditions on the unit square (see [6, §III.3], [9, §III.1]). It has the following arguments none being mandatory:

methods: a vector of strings specifying the chosen methods, available methods are:
LU: sparse LU-factorization,
CG: the conjugate gradient algorithm,
PCG: the preconditioned conjugate gradient algorithm with ssor preconditioning,
GS: gauss-seidel iteration,

SSOR: ssor iteration,
default setting is to use all methods,
nfirst: number of interior grid points in each direction on the coarsest grid, default is 3,
maxref: number of uniform mesh refinements each refinement step doubling the number of grid points in each direction, default is 5
rtol: relative error tolerance ε for iterative solvers, iteration stops once $\frac{\|Ax_n-b\|}{\|Ax_0-b\|} \leq \varepsilon$, default is 0.01,
om: relaxation parameter for ssor iteration, default is 1.5.

Note that `compare_solvers` may be called with an empty argument list and that the strings identifying methods are case sensitive. The function `compare_solvers` prints and plots the computing times of the chosen methods and their complexity defined as the ratio of the logarithm of the computing time over the logarithm of the number of unknowns.

4.9. **Eigenvalue problems.** `Numerics` provides the functions

`power_iteration`, `rayleigh`,
`inverse_power_iteration`, `inverse_rayleigh`

for computing the maximal and minimal eigenvalues of a general quadratic matrix A or of a symmetric positive definite matrix.
The functions `power_iteration` and `inverse_power_iteration` implement the power iteration (see [5, Algorithmus V.3.1], [8, §V.2]) and the inverse power iteration (see [5, Algorithmus V.3.9], [8, §V.4]) for computing the in absolute value largest and smallest, resp. eigenvalue of a general quadratic matrix A and a corresponding eigenvector. They both have the following arguments the first one being mandatory:

`A:` matrix A,
`x:` initial guess for an eigenvector, default is a random vector,
`prl:` print level, default is 0,
(0) print number of required iterations and final approximation for the largest resp. smallest eigenvalue,
(1) print number of actual iteration and actual approximation for the largest resp. smallest eigenvalue,
(2) same as 0, additionally print the final approximation of the eigenvector,
(3) same as 1, additionally print the actual approximation of the eigenvector,
`tol:` error tolerance ε, default is `1D`-8,
`nmax:` maximal number N of iterations, default is 100
iteration is stopped if two consecutive eigenvalue approximations differ less than ε and after N iterations latest.

The functions `rayleigh` and `inverse_rayleigh` realize the Rayleigh quotient iteration (see [5, Algorithmus V.3.5], [8, §V.3]) and the inverse Rayleigh quotient iteration (see [5, Algorithmus V.3.11], [8, §V.5]) for computing the largest and smallest, resp. eigenvalue of a symmetric positive definite matrix A. They both have the following arguments the first one being mandatory:

- `A:` matrix A,
- `x:` initial guess for an eigenvector, default is a random vector,
- `prl:` if `%t` print iteration number and actual approximation for the largest resp. smallest eigenvalue, otherwise print this information only at the end of the algorithm; default is `%f`
- `tol:` error tolerance ε, default is 1D−8,
- `nmax:` maximal number N of iterations, default is 100
  iteration is stopped if two consecutive eigenvalue approximations differ less than ε and after N iterations latest.

4.10. **Initial value problems for odes: fixed step-size methods.** `Numerics` provides the functions

`explicit_euler_fs`, `implicit_euler_fs`, `crank_nicolson_fs`
`rk_fs`, `rkf_fs`, `sdirk_fs`,
`nystroem_fs`, `adams_bashforth_fs`, `adams_moulton_fs`, `bdf_fs`,
`ode_fs`

for the solution of initial value problems for ordinary differential equations in $\mathbb{R}^n$

$$x'(t) = f(t, x(t)) \quad t_0 < t \leq T$$
$$x(t_0) = x_0$$

with fixed step-size methods. All functions plot the computed solution according to their argument `pc`.
The function `explicit_euler_fs` implements the explicit Euler scheme (see [6, Algorithmus I.2.1], [8, §IV.1], [10, §I.2.2]). It has the following arguments the first four being mandatory:

- `f:` force function f,
- `x0:` initial value x_0,
- `T:` final time T,
- `nt:` number of time steps, the step-size is $h = \frac{T-t_0}{\mathtt{nt}}$,
- `t0:` initial time t_0, default is $t_0 = 0$,
- `pc:` integer vector of length 2 fixing the components of x that will be plotted, default settings are:
  - `[0,1]:` for scalar odes, i.e. plot $x(t)$ versus t,
  - `[1,2]:` for systems of odes, i.e. plot the curve $(x_1(t), x_2(t))$.

Note that the user-defined function `f` must have the two arguments `t` and `x` (in this order!).
The function `implicit_euler_fs` implements the implicit Euler scheme

(see [6, Algorithmus I.2.2], [8, §IV.1], [10, §I.2.2]). It has the following arguments the first five being mandatory:

- `f:` force function f,
- `df:` derivative $D_x f$ w.r.t. x of the force function f,
- `x0:` initial value x_0,
- `T:` final time T,
- `nt:` number of time steps, the step-size is $h = \frac{T-t_0}{\texttt{nt}}$,
- `t0:` initial time t_0, default is $t_0 = 0$,
- `pc:` integer vector of length 2 fixing the components of x that will be plotted, default settings are:
  - `[0,1]:` for scalar odes, i.e. plot $x(t)$ versus t,
  - `[1,2]:` for systems of odes, i.e. plot the curve $(x_1(t), x_2(t))$.

Note that the user-defined functions `f` and `df` must have the two arguments `t` and `x` (in this order!).
The function `crank_nicolson_fs` implements the Crank-Nicolson method (see [6, Algorithmus I.2.3], [8, §IV.1], [10, §I.2.2]). It has the same arguments as `implicit_euler_fs`.
The function `rk_fs` implements the classical Runge-Kutta scheme (see [6, §I.2], [8, §IV.6.4], [10, §I.2.3]). It has the same arguments as the function `explicit_euler_fs`.
The function `rkf_fs` implements Runge-Kutta-Fehlberg schemes of order 2 with 3 stages, of order 3 with 4 stages and of order 4 with 6 stages (see [6, Beispiel I.4.7], [8, Beispiel VI.7]). It has the same arguments as the function `explicit_euler_fs` except that `pc` now is the seventh argument and that a new sixth argument `order` determines the method. This argument may take the values 2, 3 and 4 with 3 being its default value.
The function `sdirk_fs` implements a strongly diagonally implicit Runge-Kutta scheme of order 4 (see [8, §VI.4], [10, §I.2.4]). It has the same arguments as `implicit_euler_fs`.
The function `nystroem_fs` implements the Nyström two-step method (see [6, Beispiel I.5.1]). The first two approximations are computed with the explicit Euler scheme. The function `nystroem_fs` has the same arguments as `explicit_euler_fs`.
The function `adams_bashforth_fs` implements the Adams-Bashforth two- and three-step schemes (see [6, Beispiel I.5.1]). The first two or three, resp. steps are computed with the explicit Euler scheme. The function `adams_bashforth_fs` has the same arguments as the function `explicit_euler_fs` except that `pc` now is the seventh argument and that a new sixth argument `steps` determines the method. This argument may take the values 2 or 3 with 2 being its default value.
The function `adams_moulton_fs` implements the Adams-Moulton two- and three-step schemes (see [6, Beispiel I.5.1]). The first two or three, resp. steps are computed with the implicit Euler scheme. The function

`adams_moulton_fs` has the same arguments as `implicit_euler_fs` except that `pc` now is the eighth argument and that a new seventh argument `steps` determines the method. This argument may take the values 2 or 3 with 2 being its default value.
The function `bdf_fs` implements the backward difference two- and three-step schemes (see [6, Beispiel I.5.2], [8, §VI.8]). The first two or three, resp. steps are computed with the implicit Euler scheme. The function `bdf_fs` has the same arguments as `implicit_euler_fs` except that `pc` now is the eighth argument and that a new seventh argument `steps` determines the method. This argument may take the values 2 or 3 with 2 being its default value.
The function `ode_fs` allows a comparison of the above methods. Its first argument is an array `sm` of strings which identify the methods that should be compared and which may take the following values:

`"ee"`: for the explicit Euler scheme,
`"ie"`: for the implicit Euler scheme,
`"cn"`: for the Crank-Nicolson scheme,
`"rk"`: for the classical Runge-Kutta scheme,
`"rkf2"`: for the Runge-Kutta-Fehlberg method of order 2,
`"rkf3"`: for the Runge-Kutta-Fehlberg method of order 3,
`"rkf4"`: for the Runge-Kutta-Fehlberg method of order 4,
`"sdirk"`: for the strongly diagonal implicit Runge-Kutta method of order 4,
`"ny"`: for the Nystroem method,
`"ab2"`: for the Adams-Bashforth two-step method,
`"ab3"`: for the Adams-Bashforth three-step method,
`"am2"`: for the Adams-Moulton two-step method,
`"am3"`: for the Adams-Moulton three-step method,
`"bdf2"`: for the backward difference two-step method,
`"bdf3"`: for the backward difference three-step method.

The remaining arguments of `ode_fs` are the same as for the function `implicit_euler_fs`. Note that the argument `df` is also present for explicit methods as the explicit Euler or Adams-Bashforth schemes. When comparing explicit schemes exclusively you may enter the void argument [ ] for `df`.

4.11. **Initial value problems for odes: variable step-size methods.** `Numerics` provides the functions

`explicit_euler_vs`, `implicit_euler_vs`, `crank_nicolson_vs`
`rk_vs`, `rkf_vs`, `sdirk_vs`,
`ode_vs`

for the solution of initial value problems for ordinary differential equations in $\mathbb{R}^n$

$$x'(t) = f(t, x(t)) \quad t_0 < t \leq T$$
$$x(t_0) = x_0$$

with variable step-size methods. The function `rkf_vs` controls the step-size by comparing methods of different order (see [6, Algorithmus I.4.6], [8, Algorithmus VI.5]); the other functions perform the step-size control by comparing results for different step-sizes (see [6, Algorithmus I.4.3], [8, Algorithmus VI.6]). All functions plot the computed solution according to their argument `pc`.
The function `explicit_euler_vs` implements the explicit Euler scheme (see [6, Algorithmus I.2.1], [8, §IV.1], [10, §I.2.2]). It has the following arguments the first four being mandatory:

`f:` force function f,
`x0:` initial value x_0,
`T:` final time T,
`dt:` initial step-size,
`t0:` initial time t_0, default is $t_0 = 0$,
`tol:` required tolerance ε,
the function strives at obtaining an approximation $\eta(t)$ for the solution $x(t)$ such that $\|\eta(t) - x(t)\| \leq \varepsilon \|x(t)\|$ holds for all t, default value is $\varepsilon = 0.001$,
`ntmax:` maximal number of time steps, default is 10000,
`sfdt:` safety factor for step-size variation,
reduce the step-size at most by the factor $\frac{1}{\texttt{sfdt}}$ and increase it at most by the factor `sfdt`,
default value is 10,
`maxr:` maximal number of step-size reductions in a single time step, default is 10,
`pc:` integer vector of length 2 fixing the components of x that will be plotted, default settings are:
`[0,1]:` for scalar odes, i.e. plot $x(t)$ versus t,
`[1,2]:` for systems of odes, i.e. plot the curve $(x_1(t), x_2(t))$.

Note that the user-defined function `f` must have the two arguments `t` and `x` (in this order!).
The function `implicit_euler_vs` implements the implicit Euler scheme (see [6, Algorithmus I.2.2], [8, §IV.1], [10, §I.2.2]). It has the following arguments the first five being mandatory:

`f:` force function f,
`df:` derivative $D_x f$ w.r.t. x of the force function f,
`x0:` initial value x_0,
`T:` final time T,
`dt:` initial step-size,
`t0:` initial time t_0, default is $t_0 = 0$,
`tol:` required tolerance ε,
the function strives at obtaining an approximation $\eta(t)$ for the solution $x(t)$ such that $\|\eta(t) - x(t)\| \leq \varepsilon \|x(t)\|$ holds for all t, default value is $\varepsilon = 0.001$,

`ntmax:` maximal number of time steps, default is 10000,
`sfdt:` safety factor for step-size variation,
reduce the step-size at most by the factor $\frac{1}{\texttt{sfdt}}$ and increase it at most by the factor `sfdt`,
default value is 10,
`maxr:` maximal number of step-size reductions in a single time step, default is 10,
`pc:` integer vector of length 2 fixing the components of x that will be plotted, default settings are:
`[0,1]:` for scalar odes, i.e. plot $x(t)$ versus t,
`[1,2]:` for systems of odes, i.e. plot the curve $(x_1(t), x_2(t))$.

Note that the user-defined functions `f` and `df` must have the two arguments `t` and `x` (in this order!).
The function `crank_nicolson_vs` implements the Crank-Nicolson method (see [6, Algorithmus I.2.3], [8, §IV.1], [10, §I.2.2]). It has the same arguments as `implicit_euler_vs`.
The function `rk_vs` implements the classical Runge-Kutta scheme (see [6, §I.2], [8, §IV.6.4], [10, §I.2.3]). It has the same arguments as the function `explicit_euler_vs`.
The function `rkf_vs` implements Runge-Kutta-Fehlberg schemes of order 2 with 3 stages, of order 3 with 4 stages and of order 4 with 6 stages (see [6, Beispiel I.4.7], [8, Beispiel VI.7]). It has the same arguments as the function `explicit_euler_vs` except that `pc` now is the eleventh argument and that a new tenth argument `order` determines the method. This argument may take the values 2, 3 and 4 with 3 being its default value.
The function `sdirk_vs` implements a strongly diagonally implicit Runge-Kutta scheme of order 4 (see [8, §VI.4], [10, §I.2.4]). It has the same arguments as `implicit_euler_vs`.
The function `ode_vs` allows a comparison of the above methods. Its first argument is an array `sm` of strings which identify the methods that should be compared and which may take the following values:

`"ee":` for the explicit Euler scheme,
`"ie":` for the implicit Euler scheme,
`"cn":` for the Crank-Nicolson scheme,
`"rk":` for the classical Runge-Kutta scheme,
`"rkf2":` for the Runge-Kutta-Fehlberg method of order 2,
`"rkf3":` for the Runge-Kutta-Fehlberg method of order 3,
`"rkf4":` for the Runge-Kutta-Fehlberg method of order 4,
`"sdirk":` for the strongly diagonal implicit Runge-Kutta method of order 4.

The remaining arguments of `ode_vs` are the same as for the function `implicit_euler_vs`. Note that the argument `df` is also present for

explicit methods as the explicit Euler scheme. When comparing explicit schemes exclusively you may enter the void argument [ ] for `df`.

4.12. **Boundary value problems for odes.** `Numerics` provides the function

`sturm_FD`

for solving the Sturm-Liouville problem

$$\begin{aligned} -(pu')' + qu &= f \quad \text{in } (a,b) \\ u(a) &= \alpha \\ u(b) &= \beta \end{aligned}$$

with a finite difference discretization on a uniform mesh with mesh-size h (see [6, §II.4], [9, §I.6], [10, §I.5]). It has the following arguments the first two being mandatory:

`nx:` number of interior points, the mesh-size is $h = \frac{1}{\mathtt{nx}+1}$,
`f:` force function f,
`p:` diffusion function p, default is $p = 1$,
`q:` reaction function q, default is $q = 0$,
`uex:` exact solution u if known, default is [ ] signifying an unknown exact solution,
`a:` left end point a, default is $a = 0$,
`b:` right end point b, default is $b = 1$,
`al:` left boundary value α, default is $\alpha = 0$,
`be:` right boundary value β, default is $\beta = 0$.

The function `sturm_FD` plots the discrete solution and additionally the exact solution and the error if `uex` is provided. In this case `sturm_FD` also prints the L^2- and H^1-norms of the error.

4.13. **Finite difference methods for pdes.** `Numerics` provides the functions

`poisson_FD`, `heat_FD_1D`, `lax_friedrichs`, `lax_wendroff`

for finite difference discretizations of partial differential equations.
The function `poisson_FD` implements the standard 5-point difference discretization of the two dimensional Poisson equation with homogeneous Dirichlet boundary conditions

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma \end{aligned}$$

(see [6, §III.3], [9, §III.1]). The domain Ω is the intersection of the rectangle $[x_{l1}, x_{r1}] \times [x_{l2}, x_{r2}]$ with bottom-left corner x_l and top-right corner x_r with the set $\{x \in \mathbb{R}^2 : \omega(x) < 0\}$ the square $[-1,1]^2$ being the default setting. The function `poisson_FD` plots the level lines of the discrete solution and its maximal error if the exact solution is known. It has the following arguments the first two being mandatory:

`f:` force function f,

`np`: number of interior grid points in each coordinate direction, the mesh-size in x- and y-direction is $\frac{x_{r1}-x_{l1}}{\mathtt{np}+1}$ and $\frac{x_{r2}-x_{l2}}{\mathtt{np}+1}$, resp.,
`uex`: exact solution if known, default setting is [] signifying an unknown exact solution,
`xbl`: bottom-left corner x_l, default is $x_l = (-1,-1)$,
`xtr`: top-right corner x_r, default is $x_r = (1,1)$,
`fom`: function ω describing the domain Ω, default is $\omega(x) = \|x\|_\infty - 1$,
`nll`: number of level lines for plotting the solution, default is 10.

The function `heat_FD_1D` solves the one dimensional heat equation

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial^2 u}{\partial x^2} + f && \text{in } (0,T)\times(0,1) \\ u(t,0) &= 0 && \text{in } (0,T) \\ u(t,1) &= 0 && \text{in } (0,T) \\ u(0,x) &= u_0 && \text{in } (0,1) \end{aligned}$$

using a second order symmetric difference discretization for the spatial derivative and the θ-scheme for the time discretization (see [6, §III.4], [9, §III.2]). It plots the discrete solution at each intermediate time-step and has the following arguments the first three being mandatory:

`T`: final time T,
`nt`: number of time steps, the fixed time step-size is $\tau = \frac{T}{\mathtt{nt}}$,
`nx`: number of interior grid points in x direction, the fixed spatial mesh-size is $h = \frac{1}{\mathtt{nx}+1}$,
`theta`: parameter θ for the temporal discretization,
$\theta = 0$ is the explicit Eule scheme,
$\theta = 0.5$ is the Crank-Nicolson scheme,
$\theta = 1$ is the implicit Euler scheme,
default is $\theta = 0.5$,
`u0`: initial function u_0, default is $u_0 = 0$,
`f`: force function f, default is $f = 0$.

The functions `lax_friedrichs` and `lax_wendroff` implement the Lax-Friedrichs [3, Exercise 14.4] and Lax-Wendroff [3, Equation (14.29)] finite difference schemes for the numerical solution of first order partial differential equations of the form

$$\begin{aligned} \frac{\partial u}{\partial t} + a(x)\frac{\partial u}{\partial x} &= 0 && \text{in } (0,T)\times(0,1) \\ u(t,0) &= u(t,1) && \text{in } (0,T) \\ u(0,x) &= u_0 && \text{in } (0,1). \end{aligned}$$

They both plot the numerical solution at each time step and have the following arguments the first three being mandatory:

`T`: final time T,

`nx`: number of interior grid points in x direction,
the fixed spatial mesh-size is $h = \frac{1}{\texttt{nx}+1}$,
the number of time steps is automatically set to
$\texttt{nt} = \lceil T * (\texttt{nx}+1) \rceil$ which result in a constant time step $\tau \lesssim h$,
`u0`: initial function u_0,
`a`: advection function a, default is $a = 1$,
`rp`: clear plot window every `rp` time steps, default is 20.

4.14. **Linear optimization.** `Numerics` provides the functions

`simplex`, `autostart_simplex`, `dual_simplex`, `inner_point`

for linear optimization problems

$$\text{minimize } c^t x \text{ subject to } Ax = b,\ x \geq 0.$$

All functions print values of the objective function $c^t x$ and of the solution vector x depending on their argument `prl`.
The function `simplex` implements the basic simplex method (see [7, Algorithmus I.4.5]). It has the following arguments the first three being mandatory:

`clp`: cost vector $c^t \in \mathbb{R}^n$ (`clp` should be a row-vector!),
`Alp`: constraint matrix $A \in \mathbb{R}^{m \times n}$,
`blp`: constraint vector $b \in \mathbb{R}^m$ (`blp` should be a column-vector!)
`Jlp`: first basis J, default setting is $J = \{n-m+1, \ldots, n\}$.
`prl`: print level determines the print output, default is 2
(0) no print output,
(1) print optimal value $c^t x$ at the end,
(2) print optimal value $c^t x$ and vector x at the end,
(3) print value $c^t x$ at every iteration,
(4) print value $c^t x$ and vector x at every iteration,
(5) print value $c^t x$, vector x and simplex tableau at every iteration.

The function `autostart_simplex` implements the simplex method with automatic initialization (see [7, Algorithmus I.4.14]). It has the following arguments the first three being mandatory:

`clp`: cost vector $c^t \in \mathbb{R}^n$ (`clp` should be a row-vector!),
`Alp`: constraint matrix $A \in \mathbb{R}^{m \times n}$,
`blp`: constraint vector $b \in \mathbb{R}^m$ (`blp` should be a column-vector!)
`prl`: print level determines the print output, default is 2
(0) no print output,
(1) print optimal value $c^t x$ at the end,
(2) print optimal value $c^t x$ and vector x at the end,
(3) print value $c^t x$ at every iteration,
(4) print value $c^t x$ and vector x at every iteration,
(5) print value $c^t x$, vector x and simplex tableau at every iteration.

The function `dual_simplex` implements the dual simplex method (see [7, Algorithmus I.5.5]). It has the following arguments the first three being mandatory:

`clp`: cost vector $c^t \in \mathbb{R}^n$ (`clp` should be a row-vector!),
`Alp`: constraint matrix $A \in \mathbb{R}^{m\times n}$,
`blp`: constraint vector $b \in \mathbb{R}^m$ (`blp` should be a column-vector!)
`Jlp`: first basis J, default setting is $J = \{n-m+1, \dots, n\}$.
`prl`: print level determines the print output, default is 2
(0) no print output,
(1) print optimal value $c^t x$ at the end,
(2) print optimal value $c^t x$ and vector x at the end,
(3) print value $c^t x$ at every iteration,
(4) print value $c^t x$ and vector x at every iteration,
(5) print value $c^t x$, vector x and simplex tableau at every iteration.

The function `inner_point` implements an inner point method for linear optimization (see [7, Algorithmus I.7.7]). It has the following arguments the first three being mandatory:

`clp`: cost vector $c^t \in \mathbb{R}^n$ (`clp` should be a row-vector!),
`Alp`: constraint matrix $A \in \mathbb{R}^{m\times n}$,
`blp`: constraint vector $b \in \mathbb{R}^m$ (`blp` should be a column-vector!)
`xys`: initial column-vector of the form $[x; y; s]$ satisfying

$$Ax = b, \quad A^t y + s = c, \quad x > 0, \quad s > 0,$$

default setting is `xys` = [ ], `inner_point` then tries to automatically determine vectors x, y and s with the above properties,
`prl`: print level determines the print output, default is 2
(0) no print output,
(1) print optimal value $c^t x$ at the end,
(2) print optimal value $c^t x$ and vector x at the end,
(3) print value $c^t x$ at every iteration,
(4) print value $c^t x$ and vector x at every iteration,
`exact`: : default is `%f`,
if `%t`, `inner_point` tries to compute the exact solution of the optimization problem by guessing an admissible basis from the computed approximate solution and starting a simplex algorithm with this basis,
`itm`: maximal number of iterations, default is 1000,
`tol`: tolerance ε, default is $\varepsilon = 0.001$,
the iteration is stopped if the relative change of two consecutive x-vectors in the maximum norm is less than ε or if the parameter μ of inner-point method is less than ε.

4.15. **Discrete optimization.** `Numerics` provides the functions

`branch_and_bound`, `cutting_planes`,

`dijkstra`, `floyd_warshall`,
`ford_fulkerson`, `minimal_cost_flow`

for discrete optimization.
The functions `branch_and_bound` and `cutting_planes` implement branch-and-bound and cutting-planes methods, resp. for integer optimization problems

$$\text{minimize } c^t x \text{ subject to } x \in \mathbb{Z}^n,\ Ax = b,\ x \geq 0$$

(see [7, Algorithmen II.1.8, II.1.12]). They print results according to their argument `prl` and have the following arguments the first three being mandatory:

`clp`: cost vector $c^t \in \mathbb{R}^n$ (`clp` should be a row-vector!),
`Alp`: constraint matrix $A \in \mathbb{R}^{m \times n}$,
`blp`: constraint vector $b \in \mathbb{R}^m$ (`blp` should be a column-vector!)
`prl`: print level determines the print output, default is 2
(0) no print output,
(1) print optimal value $c^t x$ at the end,
(2) print optimal value $c^t x$ and vector x at the end,
(3) print value $c^t x$ at every branch-and-bound or cutting-planes iteration,
(4) print value $c^t x$ and vector x at every branch-and-bound or cutting-planes iteration,
(5) print value $c^t x$ at every simplex iteration,
(6) print value $c^t x$ and vector x at every simplex iteration.

The function `dijkstra` implements the Dijkstra algorithm for finding a shortest (s,t)-path in a graph (see [7, Algorithmus II.3.3]). It prints results according to its argument `prl` and has the following arguments the first one being mandatory:

`Adjl`: one of the following two objects:
- a square matrix such that `Adjl[i,j]` gives the length of the arc connecting nodes `i` and `j` of the graph and equals `%inf` if nodes `i` and `j` are not connected by an arc or if `i = j`,
- a list of vectors `[a,b,c]` describing the properties of the arcs such that `c` is the length of the arc connecting nodes `a` and `b`,

`first`: number of starting node s, default is 1,
`last`: number of terminal node t, default is `size(Adjl,"c")`,
`prl`: print level, default is 2,
(0) no print output,
(1) print length of shortest path at end,
(2) print length of shortest path and path itself at end,
(3) print all intermediate values of vectors D, V and M (see [7, Algorithmus II.3.3]).

The function `floyd_warshall` implements the Floyd-Warshall algorithm for finding the shortest paths between all pairs of nodes in a graph (see [7, Algorithmus II.3.11]). It prints results according to its argument `prl` and has the following arguments the first one being mandatory:

`Adjl:` one of the following two objects:
- a square matrix such that `Adjl[i,j]` gives the length of the arc connecting nodes `i` and `j` of the graph and equals `%inf` if nodes `i` and `j` are not connected by an arc or if `i` = `j`,
- a list of vectors `[a,b,c]` describing the properties of the arcs such that `c` is the length of the arc connecting nodes `a` and `b`,

`prl:` print level, default is 2,
(0) no print output,
(1) print matrix of path lengths,
(2) print matrices of path lengths and predecessors on shortest paths.

The function `ford_fulkerson` implements the Ford-Fulkerson algorithm for finding a maximal (s,t)-flow x in a network (see [7, Algorithmus II.4.6]). It prints results according to its argument `prl` and has the following arguments the first one being mandatory:

`Adjc:` one of the following two objects:
- a square matrix such that `Adjc[i,j]` gives the capacity of the arc connecting nodes `i` and `j` of the graph and equals `-%inf` if nodes `i` and `j` are not connected by an arc or if `i` = `j`,
- a list of vectors `[a,b,c]` describing the properties of the arcs such that `c` is the capacity of the arc connecting nodes `a` and `b`,

`first:` number of starting node s, default is 1,
`last:` number of terminal node t, default is `size(Adjc,"c")`,
`prl:` print level, default is 2,
(0) no print output,
(1) print the value of the maximal flow x,
(2) print the value of the maximal flow x and the corresponding matrix where the element `[i,j]` gives the flow through the arc connecting nodes `i` and `j`,
(3) print in every iteration the value of the current flow x,
(4) print in every iteration the value of the current flow x and the corresponding matrix where the element `[i,j]` gives the flow through the arc connecting nodes `i` and `j`.

The function `minimal_cost_flow` implements algorithm II.4.16 of [7] for finding an (s,t)-flow x with prescribed value Φ and minimal cost in

a network. It prints results according to its argument `prl` and has the following arguments the first three being mandatory:

`Adjc`: one of the following two objects:
- a square matrix such that `Adjc[i,j]` gives the capacity of the arc connecting nodes `i` and `j` of the graph and equals `-%inf` if nodes `i` and `j` are not connected by an arc or if `i = j`,
- a list of vectors `[a,b,c]` describing the properties of the arcs such that `c` is the capacity of the arc connecting nodes `a` and `b`,

`Adjp`: one of the following two objects:
- a square matrix such that `Adjp[i,j]` gives the cost of the arc connecting nodes `i` and `j` of the graph and equals `%inf` if nodes `i` and `j` are not connected by an arc or if `i = j`,
- a list of vectors `[a,b,c]` describing the properties of the arcs such that `c` is the cost of the arc connecting nodes `a` and `b`,

`val`: requested value Φ of the flow,
`first`: number of starting node s, default is 1,
`last`: number of terminal node t, default is `size(Adjc,"c")`,
`prl`: print level, default is 2,
(0) no print output,
(1) print the value of the final flow x,
(2) print the value of the final flow x and the corresponding matrix where the element `[i,j]` gives the flow through the arc connecting nodes `i` and `j`,
(3) print in every iteration the value of the current flow x,
(4) print in every iteration the value of the current flow x and the corresponding matrix where the element `[i,j]` gives the flow through the arc connecting nodes `i` and `j`.

4.16. **Nonlinear optimization.** `Numerics` provides the functions

`descent`, `augmented_lagrangian`, `nelder_mead`

for nonlinear optimization.
The function `descent` realizes a descent algorithm for unconstrained nonlinear optimization problems

$$\text{minimize } f(x) \text{ in } \mathbb{R}^n$$

(see [7, Algorithmus III.1.5]). The search direction is the negative gradient of the objective function f. The line search is either an exact one or an Armijo one with parameters $c_1 = c_2 = 0.5$ (see [7, Korollar III.1.10]). In the first case the user must provide the Hessian matrix $D^2 f$ of the objective function. The function `descent` prints results according to its argument `prl` and has the following arguments the first three being mandatory:

`xp`: initial guess x_0,
`fct`: objective function f,
`gfct`: gradient Df of f (`gfct` should be a column-vector!),
`Hfct`: Hessian D^2f of f,
if `Hfct` is provided the line search is an exact one otherwise it is an Armijo one,
default is `Hfct` $= [\,]$, i.e. Armijo line search,
`prl`: print level, default is 2,
(0) print optimal value of f,
(1) print optimal value of f and corresponding point x,
(2) print current value of f at every iteration,
(3) print current value of f and x at every iteration,
`tol`: tolerance ε, default is $\varepsilon = 1.0\mathtt{D}{-}4$,
`itmax`: maximal number N of iterations, default is $N = 100$,
iteration terminates if $\|Df(x)\| \le \varepsilon$ or if the number of iterations exceeds N.

The function `augmented_lagrangian` implements an augmented Lagrangian algorithm for constrained nonlinear optimization problems

$$\begin{aligned} &\text{minimize } f(x) \text{ subject to} \\ &\qquad f_i(x) \le 0, \quad 1 \le i \le p, \\ &\qquad f_j(x) = 0, \quad p+1 \le j \le m \end{aligned}$$

with augmented Lagrange function

$$\begin{aligned} \Lambda(x,y,r) = f(x) &+ \sum_{i=1}^{p} \frac{1}{2} r_i \left[\left(f_i(x) + \frac{y_i}{r_i} \right)^+ \right]^2 \\ &+ \sum_{j=p+1}^{m} \frac{1}{2} r_j \left[f_j(x) + \frac{y_j}{r_j} \right]^2 \\ &- \sum_{k=1}^{m} \frac{1}{2} \frac{y_k^2}{r_k} \end{aligned}$$

(see [7, Algorithmus III.6.9]). It prints results according to its argument `prl` and has the following arguments the first eight being mandatory:

`xp`: initial guess for the vector x,
`yp`: initial guess for the vector y,
`rp`: augmentation factor,
the vector r is given by $r =$ `rp` $(1, \dots, 1)^t$,
`fct`: objective function f,
`gfct`: gradient Df of f (`gfct` should be a column-vector!),
`fctc`: vector valued constraint function $(f_1, \dots, f_m)^t$
(`fctc` should be a column-vector!),
`gfctc`: gradient of `fctc` (gradients of the component functions of `fctc` should be column-vectors!),

`nic`: number p of inequality constraints,
the values $p = 0$, no inequality constraints, and $p = m$, no equality constraints, are allowed,
the first `nic` components of the vector `yp` must be positive,
`prl`: print level, default is 2,
(0) print optimal value of f,
(1) print optimal value of f and corresponding point x,
(2) print current value of f at every iteration,
(3) print current value of f and x at every iteration,
`tol`: tolerance ε, default is $\varepsilon = 1.0\mathtt{D}{-}4$,
`itmax`: maximal number N of iterations, default is $N = 100$,
iteration terminates if the KKT-conditions are satisfied up to tolerance ε or if the number of iterations exceeds N.

The function `nelder_mead` implements the simplex method of Nelder and Mead for unconstrained nonlinear optimization problems

$$\text{minimize } f(x) \text{ in } \mathbb{R}^n$$

(see [7, Algorithmus III.8.1]). It prints results according to its argument `prl` and has the following arguments the first two being mandatory:

`fct`: objective function f,
`inx`: either the dimension n of the problem or an $n \times (n+1)$ matrix with the initial guesses $x_0, \ldots, x_n$ as column vectors,
in the first case the initial guesses are automatically set to the origin $x_0 = 0$ and to the unit vectors $x_i = e_i$, $1 \le i \le n$,
`prl`: print level, default is 2,
(0) print optimal value of f,
(1) print optimal value of f and corresponding point x,
(2) print current best value of f at every iteration,
(3) print current best value of f and corresponding point x at every iteration,
`itm`: maximal number N of iterations, default is $N = 100$,
`tol`: tolerance ε, default is $\varepsilon = 0.001$,
iteration terminates if the variation of the function values is less than ε or if the number of iterations exceeds N.

References

[1] Michaël Baudin, *Introduction to Scilab.* http://wiki.scilab.org/Tutorials
[2] ———, *Programming in Scilab.* http://wiki.scilab.org/Tutorials
[3] Arieh Iserles, *A First Course in the Numerical Analysis of Differential Equations.* Cambridge Texts in Applied Mathematics, Cambridge University Press 1997
[4] Eike Rietsch, *An Introduction to Scilab from a Matlab User's Point of View.* http://wiki.scilab.org/Tutorials

[5] Rüdiger Verfürth, *Einführung in die Numerische Mathematik.*
http://www.rub.de/num1/files/lectures/EinfNumerik.pdf
[6] ———, *Numerik I. Gewöhnliche Differentialgleichungen und Differenzenverfahren für partielle Differentialgleichungen.*
http://www.rub.de/num1/files/lectures/NumDgl1.pdf
[7] ———, *Optimierung.*
http://www.rub.de/num1/files/lectures/Optimierung.pdf
[8] ———, *Numerische Mathematik für Maschinenbauer, Bauingenieure und Umwelttechniker.*
http://www.rub.de/num1/files/lectures/NumIng.pdf
[9] ———, *Vertiefung Numerische Mathematik für den Masterstudiengang UTRM.*
http://www.rub.de/num1/files/lectures/VertNum.pdf
[10] ———, *Numerical Methods and Stochastics. Part I: Numerical Methods.*
http://www.rub.de/num1/files/lectures/NMS.pdf

Index

adams_bashforth_fs, 13
adams_moulton_fs, 13
augmented_lagrangian, 24
autostart_simplex, 19

bdf_fs, 14
bicg_stab, 10
boolean, 2
branch_and_bound, 21

cg, 10
compare_solvers, 10
crank_nicolson_fs, 13
crank_nicolson_vs, 16
cubic_spline, 7
cutting_planes, 21

descent, 23
dijkstra, 21
dual_simplex, 20

explicit_euler_fs, 12
explicit_euler_vs, 15

floyd_warshall, 22
ford_fulkerson, 22
function, 4

gauss_rule, 8
gauss_seidel, 9
goertzel, 7

heat_FD_1D, 18
hermite, 6

implicit_euler_fs, 12
implicit_euler_vs, 15
inner_point, 20
inverse_power_iteration, 11
inverse_rayleigh, 12

jacobi, 9

lagrange, 6
lax_friedrichs, 18
lax_wendroff, 18

matrix, 3
midpoint_rule, 8
minimal_cost_flow, 22

nelder_mead, 25
neville, 5
newton, 8
number, 3
numerics_help, 5
nystroem_fs, 13

ode_fs, 14
ode_vs, 16

pbicg_stab_ssor, 10
pcg_ssor, 10
poisson_FD, 17
power_iteration, 11

rayleigh, 12
regula_falsi, 9
richardson, 9
rkf_fs, 13
rk_fs, 13
rkf_vs, 16
rk_vs, 16
romberg, 8

sdirk_fs, 13
sdirk_vs, 16
secant_rule, 9
simplex, 19
simpson_rule, 8
sparse matrix, 3
ssor, 9
string, 3

trapezoidal_rule, 8

vector, 3